オブジェクト指向プログラム設計記述言語OOPDとその記述環境

加藤木 和夫† 畠山 正行†† 上田 賀一††

科学技術分野のドメインユーザ向けに要求記述からプログラム製作までを日本語で一貫して記述するプログラミング環境を構築し，次いで未整備であった分析・設計段階の記述言語 OODJ の強化，改良を行ってきた．本論文では，OODJ の強化・改良にともなうプログラム設計段階の記述を大幅に見直し，よりいっそう自然日本語に近い形式での一貫性を追求した記述言語 OOPD を開発した．開発にあたってはプログラム設計記述モデル，OOPD 文法，OODJ から OOPD への変換手順等を新たに設計した．OODJ の日本語単文記述を OOPD へ変換する手順として，パターン化，メッセージ化，近似表現化のステップを明確にし，OOPD を自然言語の強力さを保ちつつ実行可能な言語仕様を持つ記述言語として確立できた．また，OOPD 記述支援環境を設計，構築し，記述実験を行った．その結果，スムーズ感の向上，生産効率向上が確認でき，高水準の一貫プログラミング環境を構築し得たと確信する．本環境によりドメインユーザはプログラム開発力を大幅に向上させることが可能である．

Object-Oriented Program Design Description Language OOPD and Its Description Environment

Kazuo Katougi,† Masayuki Hatakeyama†† and Yoshikazu Ueda††

To improve the program development ability of the domain users in various science and technology fields, we have designed a new description language called OOPD (Object-Oriented Program Design description language) based on the Natural Japanese in the program design processes. We have introduced and designed a new description model, the grammar of OOPD, and the rules to translate from the analysis process description language into the OOPD ones. To translate easily the behavior descriptions into OOPD description, we have introduced the typical sentence pattern technique, the message description technique, and the pre-statement description technique. We have succeeded in constituting the executable design description language OOPD with the powerfulness of the natural language. We have also designed and re-constituted the description support environment, and have examined the description tests. As the results, we have confirmed that the OOPD and its description environment have realized the sufficient programming tools for the domain users to describe/develop their own programs.

1. はじめに

本論文の狙いはソフトウェア非専門家のために自然日本語に近い言語を用いてプログラム記述することはできないか，という点にある．

このような狙いを持つ本論文において，第 1 の着目点は，非専門家が馴染めないプログラミング言語（Programming Language，以下必要に応じて PL と略）の持つ記号性と厳密さである．PL は出発点が英語であり，キーワードには英単語の短縮語が含まれており，非英語圏の国民にとっては意味のある言葉というよりは非親和的な記号に近い．解決法として日本語プログラミングがあげられる[1),2)]．日本語プログラミングは識別子の日本語化から自然語に類似した書き方へと進化はしてきたが，厳密な文法を持つ PL の枠は超えていない．キーワードは日本語であるがやはり記号に近い．また，プログラミング技術の習得を必要とする点では従来の PL と同じで，根本的な解決にはなっていない．

第 2 の着目点はドメインユーザからのプログラミング負担の軽減要求である．本研究が対象とするドメインユーザは科学技術計算・研究系分野のユーザである．確かに事務計算・ビジネス系分野では，比較的処

† 加藤木技術士事務所
Katougi Technical Office
†† 茨城大学工学部情報工学科
Department of Computer and Information Sciences, Faculty of Engineering, Ibaraki University

理を定型化しやすいことから表計算ソフト，第4世代言語等のプログラミングツール[3]が整備されているが，本研究の対象分野では，アルゴリズムを幾度も書き直しながら研究結果を求めていくので，定型的な記述が難しく，Fortranを中心にユーザ自身が従来の手続き的な記述を続けている．特定分野のプログラム開発環境[4),5)]はあるものの，分野や設定条件が少しでも異なると使えず，プログラミングツールの整備は遅れている．また，ユーザの主たる関心事は実験結果にあり，アルゴリズム記述のツールはほしいが，新たに厳密な文法を持つPLの習得は避けたい，という要望を強く持っている．

このような点に対し，我々はドメインユーザ向けに自然言語風の制約のある日本語でプログラムを記述する記述言語（Description Language，以下必要に応じてDLと略）の方式を提案している[6)~8)]．本方式はDLをできるだけ自動的にプログラムに変換することが研究の中心的な課題となる．DLは記述性や可読性，理解容易性の点でPLとは画然と異なる．しかし，制約を付けるとはいえ，日本語の文章に近いDL記述からのプログラム自動生成の困難性はコンパイラが自然言語の持つ多様性・曖昧性・非構造性に対応できないことから明白である．

ドメインユーザのプログラム開発環境を実現するために，我々は開発過程をいくつかのステップに分け，各々に対応したDL，すなわちオブジェクト指向自然日本語**OONJ**（Object-Oriented Natural Japanese）[13)]，オブジェクト指向記述日本語**OODJ**（Object-Oriented Description Japanese）[8)]，日本語プログラム記述言語**JSMDL**（Japanese Software Modeling Description Language）[6),7)]の複数言語を系統的に開発し，日本語一貫プログラミング環境[6)~9)]として提案・設計・構築している．

本論文では，OODJ文法[8)]の確定にともない，日本語プログラム記述言語JSMDLを**図1**のように5つのポイントから全体的に見直し，再構築する．

本論文では，第1に(1)にあるようにJSMDLに比較してDLとしての位置付けを再検討し，変更した．プログラム記述言語であったJSMDLを「プログラム設計」記述言語とすることで，OODJから円滑につながり，かつPLを直接扱うことのない設計段階のDLへの変更と再定義を行った．

JSMDLからの改訂にあたっては，PLの実行可能性を保持しつつプログラム設計記述という上流工程記述へ言語仕様を移行した．そのためにはまず日本語一貫プログラミングでの位置付けを明らかにし，次にプログラム設計記述言語としての言語仕様，前記述段階のOODJからの変換規則および処理系の全般を再構築する．

第2に，(2)設計記述モデルの新規な設計を行った．JSMDLは対象世界をどのように記述するかを表す記述モデルは特には明示してはいなかった．本論文ではこの点を再検討し，設計仕様に要求される事項を明確にし，それをモデルに反映する．それにともなって，(3)記述モデルに対応する文法を再定義する．

第3には(4)のOODJからの変換規則とその手順の見直しである．OODJでは対象世界のオブジェクト構造が明確に定義できるようになったが，振舞いの部分はまだ日本語単文の列である．この日本語単文列をプログラム設計記述言語へ変換する手順を見直し，ドメインユーザが円滑にプログラム化を図れるようにした．

最後に，上記の再検討と変更・改訂にともなう(5)の定式化と環境の実現である．まず可読性の高い日本語記述を実行可能プログラムへ変換する方式を定式化し，この方式を支援環境として実現することで，ドメインユーザが新たなPLを習得せずに現在持っているFortran程度のPL知識で，対象世界を日本語でプログラム化できるようにする．

以上のような方針で導入するプログラム設計記述言語を**OOPD**（Object-Oriented Program Design description language）と呼ぶ．本論文では，2章で現状の日本語一貫プログラミング環境の問題と課題，3章ではOOPDの設計要件の検討，4章では記述モデルとOOPD文法の設計，5章ではOODJからOOPDへの変換の課題と解決法，6章では支援環境について記述し，7章では記述実験と評価を，8，9章で評価と結果を記述する．

(1)プログラムDLからプログラム設計DLへのシフト．
(2)設計仕様を取り込んだ設計記述モデルの新規設計．
(3)記述モデルに対応し，文法を再定義．
(4)OODJからの変換規則・手順の再検討と振舞い記述の変換手順の見直し．
(5)実現目標は可読性の高さと実行可能プログラムの両立を目指す変換方式と支援環境の構築．

図1 JSMDLの再検討事項
Fig. 1 Reconsideration item of JSMDL.

2. 日本語一貫プログラミング環境の問題と課題

2.1 日本語一貫プログラミング環境

プログラムDLであるJSMDLとそのプログラミ

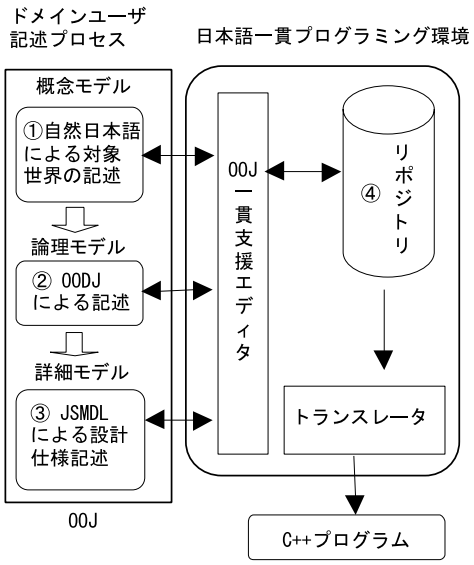


図 2 オブジェクト指向日本語一貫プログラミング環境
Fig. 2 Object-oriented, integrally consistent Japanese programming environment.

ング環境 JOMON を持つ日本語一貫プログラミング環境[7] を**図 2** に示す．図 2 において，① 概念モデルでは要求，具体的なシナリオを自然言語としての日本語（以下，自然日本語と呼ぶ）で記述する．② 次に OODJ 記述へと変換する．OODJ は自然日本語に OO 構造記述を追加し，自然日本語からの変換規則を定めた言語である．各シング（Thing：クラス相当）の振舞い（メソッド相当）記述は自然日本語に緩い制約（単文のみ許す．複文，疑問文等は変換する）を加えたもので記述する．③ OODJ 記述を JSMDL 記述へと変換する．JSMDL は C++風の日本語プログラム DL である．この記述結果はトランスレータにより C++プログラムに変換され，実行される．④ 各モデル間の情報はリポジトリに蓄積・利用される．

2.2 日本語一貫プログラミング環境の問題点

OODJ 文書を JSMDL プログラムへと，自動的に変換できる部分はクラス構造，クラス間関係であり，自動的にできない部分は振舞い記述のところである．現状ではユーザ自身が OODJ の振舞い記述を見ながら，プログラムへの変換を頭の中で考え JSMDL プログラムを記述するという従来と変わらない方法をとっている．このため，現状の日本語一貫プログラミング作業では振舞い記述に関しては OODJ と JSMDL の間に大きな隔たりが残っており，ドメインユーザのプログラミング負担は従来と変わっていないという大きな問題が残っていた．これは日本語を用いての一貫プログラミングを実現するために克服しなければならない大きい課題である．

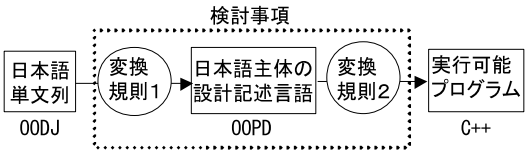


図 3 変換システム
Fig. 3 Translation system.

我々はこの課題を解決するために，設計ポイントをプログラム DL である JSMDL より上流のプログラム設計 DL へと移行し，OODJ からプログラムへの変換格差を少なくし，さらに OODJ からの変換規則を明確にすることで，ドメインユーザのプログラミング負担を可能な限りなくすこととした．ただし，上流の OODJ 方向へ移行することで，JSMDL が本来持っている実行可能性が失われないようにする．ここで，実行可能性とは記述したものが単なる文書ではなく，実行可能なプログラムであることを意味する．いい換えるとトランスレータを実現できる言語である．

重要な点はドメインユーザの負担軽減の実現にあたり，OODJ 文書の高可読性を維持しつつ JSMDL の実行可能性を実現することである．

また，ドメインユーザは従来の一貫プログラミングのプロセスにおいて，論理モデル（シング単位）を JSMDL（クラス単位）へ単純に変換するだけであって，必要な設計方法や手順を用いない傾向があり，これが設計全体の見通しを悪くしている，という現状がある．これに対し我々は設計仕様の記述モデルを明確にし，内部に設計手順を取り込むことでドメインユーザが設計全体をよく見通せるようにし，設計しやすくすることとした．

設計記述言語 OOPD を導入した後の OODJ から C++に至る変換システムは**図 3** のようになる．なお，ここで変換規則はユーザの手作業による変換のための規則，エディタ/トランスレータといった環境を包括する概念である．変換規則 1 は OODJ から OOPD への変換，変換規則 2 は OOPD から C++への変換規則とする．

2.3 設計課題

設計課題は**図 4** のように構成される．OODJ のように厳密な振舞い記述構文を持たない日本語記述に対し，変換規則 1 を適用して可能な限り OOPD に近いところまで変換する．OOPD はプログラム設計の記述実現が重要なポイントとなる．最後の変換規則 2 は既存の JSMDL トランスレータ[6),7)]（C++コード生

分類	変換規則1	設計記述言語	変換規則2
設計項目	変換規則，手順	OOPD文法	C++コード生成規則
内容	OODJで振る舞いはどこまで記述可能かを検証し，OODJ記述をOOPDに変換する規則を確立する．	JSMDLの文法を（実行可能な）設計記述言語OOPDの文法として見直す．	JSMDLの文法見直しに伴い既存のJSMDLトランスレータを再設計する．

図 4 設計課題
Fig. 4 Design problem.

成規則）の改造を行う．

3. OOPD の設計要件の検討

3.1 OOPD の位置付けと特長

プログラム設計 DL として JSMDL（2 章参照）の改訂版 OOPD を定義する．OOPD はプログラム設計記述を目的とし，変換規則 2 により実行可能なプログラムへ変換できる DL とする．本言語は**図 5** に示すように OODJ の PL 寄りに，JSMDL を包含するように位置付けられる．図 5 において OOJ（Object-oriented Japanese）は言語系であるオブジェクト指向日本語一貫 DL の総称である．

図 5 からも分かるように OODJ の単文は表現の自由度が高い代わりに一意性はなく，実行可能プログラムとはならない．OOPD 文法は OODJ 記述を OOPD 記述の第 1 ステップとし，変換規則 1 により順次書き直していくものと位置付ける．書き換えていくうちに，一意性を備えるようになり，実行可能な OOPD 記述となるように設計する．そして最終記述の文法は変換規則 2 により実行可能なプログラムへ（自動的かつ完全に）変換可能なものとする．この最終ステップの記述を OOPD の文法とする．したがって，OOPD の第 1 ステップの記述と最終ステップ記述の間には中間ステップの記述を許す文法とする．

3.2 プログラム設計記述言語の要件

プログラム設計 DL への要件を**図 6** にまとめて示す．重要な点は図 6 の 2 項，ドメインユーザの対象世界の特質からの要件で，全体の振舞い（オブジェクト操作等）記述部分と個々のオブジェクト定義部分とに記述を明確に分けることと，図 6 の 4 項，設計記述言語からの要件で，外部仕様と内部仕様の記述を分離することである．

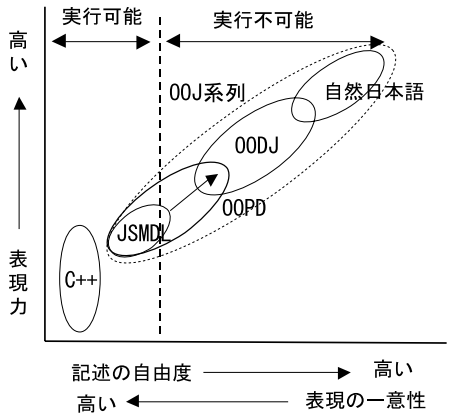


図 5 OOPD の位置付け
Fig. 5 Position of OOPD in OOJ.

1．ドメインユーザからの要求：新規PLを習得不要で現用PL(Fortranライク)の知識を前提．
2．対象世界の特質に合わせ，全体の振舞い実験単位の記述，オブジェクト操作記述部分と個々のオブジェクト定義部分を明確に分離．
3．オブジェクト指向設計（クラス，クラス継承，メッセージ交信）の実現．
4．外部仕様と内部仕様の記述分離と作業順序のガイド．外部仕様記述と内部記述とを分離記述し，外部仕様から内部仕様へ進めるという作業手順を明確にし，ユーザをガイドする．外部仕様は他クラスへ公開，内部仕様は非公開（情報隠蔽の実現）．
5．外部仕様の記述には属性名や振舞いの外部インタフェースを記述．
6．内部仕様には実行可能なアルゴリズムを記述．
7．内部仕様記述には記述段階として複数ステップが可能
8．PLへの変換機能（一意性保持）．
9．PL特有の機能削除（設計DLであるため）．

図 6 プログラム設計 DL の要件
Fig. 6 Requirement conditions for program design DL.

(1) 対象世界はクラスを記述単位とするクラス定義系と実験単位を記述単位とする操作系とに分離して記述する．
(2) オブジェクト指向設計（クラス，クラス継承，メッセージ交信等）を実現する
(3) 外部仕様と内部仕様との記述を分離する．
(4) 外部仕様には属性および外部に開放する相互作用のインタフェースを記述する．
(5) 内部仕様にはアルゴリズムを記述する．
(6) 既存のプログラム言語への変換を可能とする．

図 7 OOPD の記述モデル
Fig. 7 Description model for OOPD.

4. OOPD の記述モデルと文法の設計

4.1 記述モデル：何を記述するか

図 6 の要件を実現する記述モデルを**図 7** に示す．記

```
1．クラス名
2．外部仕様
2.1 クラス間相互関係（クラス継承指定）
      記述構成（継承クラス名リスト）
2.2 属性
      記述構成（属性名，アクセス制約，型）
      のリスト
2.3 動作
      記述構成（アクセス制約，動作名，引数
            リスト，戻り値の型）のリスト
3．内部仕様
3.1 属性の詳細定義
      記述構成（属性名，アクセス制約，型，
            属性値）
3.2 動作の詳細定義
  (1)動作インタフェース定義
      記述構成（アクセス制約，動作名，引数
          リスト，戻り値の型）
  (2)一時変数：この動作内のみ有効
      記述構成　（変数名，データ型）
  (3)動作本体
      記述構成　（文）のリスト
```

図 8 記述モデル（クラス定義系）
Fig. 8 Description model (class definition part).

```
1．操作名
2．操作の手順
2.1　一時変数
     記述構成　（変数名，型）
     この操作系内のみ有効
2.2　操作本体
     記述構成　（文）のリスト
```

図 9 記述モデル（操作系）
Fig. 9 Description model (operation part).

述モデルとして対象世界のクラスを記述単位とするクラス定義と，対象世界の実験単位を記述単位とする操作系を分離する．また，クラス定義系の中は，外部仕様と内部仕様とに分ける．この要件を満たす記述モデルとして設計されたのが**図 8** クラス定義系の記述モデルと**図 9** 操作系の記述モデルである．各系の記述項目の記述構成は ( ) で囲み示す．

図 8, 9 を補足する．図 8 の**2.2 属性**におけるアクセス制約は，共有（クラス外へ公開），限定共有（特定のクラスへのみ公開）および私有（自クラス内のみ）の 3 種類を設ける．同様に**2.3 動作**のアクセス制約としては，クラス外公開，限定公開およびクラス内部のみ有効の 3 種類を設ける．また，**3.1 属性詳細の定義**の中の型はシステムの用意した基本的な型とユーザの定義した型の 2 種類を許す．図 8 の**3.2 (3) 動作本体**と図 9 の**2.2 操作本体**の文はメッセージ送信文等である．

4.2 OOPD の文法：どう記述するか

構文の設計方針を**図 10** にまとめた．

図 11，**図 12**，**図 13** に図 10 の方針に基づいて設計した OOPD の構文を示す．図 11，図 12，図 13 の中で定義されていない非終端記号は一般の常識に従う

```
(1)外部・内部仕様記述分離による情報隠蔽の実現．
(2)属性：外部仕様では型のみを記述．属性値は内部
   仕様に記述．型は基本的なデータ型に限定．
(3)動作の仕様：動作のインタフェースは外部仕様に
   記述．内部のアルゴリズムは内部仕様に記述．
(4)メッセージ送信：相互作用はメッセージ式で記述．
     メッセージ式：　＜項＞　←　＜メッセージ＞
   相互作用の特定のために″←″を採用．
(5)制御文：繰り返しのloop文，選択のif文，多分岐の
   cond文．　見慣れた英文字採用．
(6)アクセス制約：属性名や動作名のアクセス制約は
   文法では削除．PLへの変換時に属性名はクラス内のみ
   有効な「private」，動作名は全クラスで共有できる
   「public」へ変換．
```

図 10 OOPD の設計方針
Fig. 10 Design principles for OOPD syntax.

```
〈外部仕様〉          ::= “%外部仕様”〈クラス外部仕様〉
〈クラス外部仕様〉    ::= “%クラス”〈クラス名〉
                          〈クラス間相互関係〉
                          [{〈属性外部仕様〉}][{〈動作外部仕様〉}]
〈クラス間相互関係〉::= “%クラス間相互関係”
                          [〈クラス特性〉][〈関連クラス〉]
〈クラス特性〉        ::= 〈任意の文字列〉
〈関連クラス〉        ::= 〈汎化クラス〉|
                          〈集約クラス〉|{〈関係クラス〉}
〈汎化クラス〉        ::= “is_a”〈上位クラス名〉
〈集約クラス〉        ::= “have”〈集約クラス名〉
〈関係クラス〉        ::= “assoc”〈関係クラス名〉〈関係の名前〉
〈属性外部仕様〉      ::= “%属性”〈属性名〉×〈属性の型〉
〈動作外部仕様〉      ::= “%動作”〈動作定義〉
〈動作定義〉          ::= 〈動作名〉{〈引数〉}〈戻り値の型〉
〈引数〉              ::= 〈引数名〉×〈引数の型〉
〈戻り値の型〉        ::= 〈属性の型〉
```

図 11 OOPD 構文（定義系：外部仕様）
Fig. 11 OOPD syntax (class definition part: external specification).

ものとする．

4.3 JSMDL 文法からの改訂点

プログラム設計 DL としての位置付けから，設計仕様の要件であるクラス定義と操作系記述の分離と外部仕様と内部仕様記述の分離を取り込んだ．一方，OO 設計記述と動作記述（文）については一部細かい点の変更はあるが，大筋は JSMDL を継承した．これはこの部分は JSMDL ですでに実現されているためである．

5. OODJ から OOPD への変換の課題と解決法

本章では OOPD の動作記述（文）に焦点を当て，OODJ から OOPD への変換規則 1（2.3 節および図 4）の課題とその解決策について述べる．OODJ 文法の日本語単文はプログラム化するための情報が不足しており，またアルゴリズムとしても未完の文である．以下，この OODJ 記述から OOPD 記述への変換プロ

```
〈内部仕様〉      ::= “%内部仕様”〈クラス内部仕様〉
〈クラス内部仕様〉::= “%クラス”〈クラス名〉
                    [{〈属性内部仕様〉}][{〈動作内部仕様〉}]
〈属性内部仕様〉  ::= “%属性”〈属性名〉
                    〈アクセス制限〉〈属性の型〉〈属性値〉
〈動作内部仕様〉  ::= “%動作”〈動作名〉{〈メソッド〉}
〈メソッド〉      ::=〈文の並び〉
〈文の並び〉      ::= {〈文〉“. ”} “return”〈文〉“. ”
〈文〉            ::=〈メッセージ文〉|〈loop文〉|
                    〈代入文〉|〈if文〉|〈cond文〉|
〈メッセージ文〉  ::=〈メッセージ式〉
〈loop文〉        ::= “loop”〈条件〉
                     “{” 〈文の並び〉 “}”
〈if文〉          ::= “if”〈条件〉“then”
                          “{”〈文の並び〉“}”
                       “else”    “{”〈文の並び〉“}”
                     “endif”
〈cond文〉        ::= “cond”〈条件 〉 “{”
                     { “case”〈定数式〉
                          “{”〈文の並び〉“}” }
                       “default”
                          “{”〈文の並び〉“}”
                     “}”
〈条件〉          ::= “(”〈項〉|〈式〉“)”
〈代入文〉        ::=〈変数〉“：＝”〈式〉
〈式〉            ::=〈メッセージ式〉|
                    〈２項式〉|〈単項式〉
〈メッセージ式〉  ::=〈項〉“←”〈メッセージ〉
〈２項式〉        ::=〈項〉〈２項演算子〉〈項〉
〈単項式〉        ::=〈単項演算子〉〈項〉
〈メッセージ〉    ::=〈メッセージ名〉[“(”{〈項〉“, ”} “)”]
〈項〉            ::=〈基本項〉|“(”〈式〉“)”
〈基本項〉        ::=〈属性名〉|〈リテラル〉
```

図 12 OOPD 構文（定義系：内部仕様）
Fig. 12 OOPD syntax (class definition part: internal specification).

```
〈操作〉      ::= “%操作”〈操作名〉
                 [〈一時変数〉]〈操作仕様〉
〈一時変数〉::= “%一時変数”〈一時変数名〉
                 〈属性の型〉〈属性値〉
〈操作仕様〉::= “%動作”{〈メソッド〉}
〈メソッド〉::=〈文の並び〉
〈文の並び〉::= {〈文〉“. ”}
               “stop. ”
```

図 13 OOPD 構文（操作系）
Fig. 13 OOPD syntax (operation part).

セスについて分析し，その変換の課題を明確にし，解決策となる変換規則の定式化を行う．

なお，変換規則 2 は OOPD から C++への変換，すなわち C++コード生成規則である．これは従来の JSMDL トランスレータ[7] を改造することで実現するので，詳細検討は省略する．

5.1 変換のプロセス分析と課題

変換に必要な事項を 1 つの日本語単文に沿って分析すると**図 14** のようになる．変換プロセスは**図 15** にまとめた．

日本語単文から OOPD 記述への変換プロセスは上記を 2 ステップにまとめ，データ型等の詳細を設定する 1 ステップを加えて以下の 3 ステップで完成させ，トランスレータへ入力する．

(a) 単文を品詞分解し，メッセージ形式へ書き直す．
(b) クラス名等の特定を行い，ほぼ OOPD 表記に近い形にする．
(c) データ型等の詳細を補って OOPD 表記とする．

この 3 つの変換ステップは日本語単文の品詞分解からスタートするが，ドメインユーザにとって実際には一般的な文章をオブジェクト指向のメッセージを意識して分解し，メッセージ形式に書き直すのは簡単でない．そこで，上記 (a)，(b)，(c) のステップの前にあらかじめ日本語単文の段階でオブジェクト指向を意識した文章へまず変換し（記述し直し），そのうえでメッセージ形式へ書き直す方法を検討することとした．以下 OODJ 記述に遡り，日本語単文段階でのオブジェクト指向記述への書き直しを詳細に検討する．

5.2 OODJ 記述からの新たな変換法の提案

OODJ 記述[8] では論理的には矛盾がなく，構文としても平述文と簡易制御文からなり，記述の自由度に制限をあまり加えることなくシステムの分析記述が行える．しかし自由度はまだ大きく，オブジェクト指向の基本文型であるメッセージ送信形式への変換はユーザの記述能力に左右される．特にプログラムロジックに不慣れなユーザにとっては単文記述からメッセージ送信形式へ書き直すのはなかなか面倒な作業である．

我々は自由度を制限する方法として，OODJ 単文に**図 16** のような振舞い記述に対するパターンを導入した．OODJ の自由度の高い振舞い記述に記述制限を加えるパターン化は単文の 1 つの文全体をいくつかのパターンに分類する方法と単文の中の部分に注目して分類する方法が考えられる．しかし前者の方法は意味まで考慮すると特定分野のメソッド群を作成することと同じになり，パターン化の汎用性がなくなる．また，単文といえども複雑な意味内容を含み文単位の分類ではパターン化が難しい．我々はオブジェクト指向の観点からもう一度 OODJ の基本文型を見直し，相互関係を表現する次の 5 つの典型的文型をパターンとして抽出・定型化した．図 16 にそれらを提示する．

(A) と **(B)** はオブジェクト間の静的な構造関係を表しており，相互関係ですでに表現されている．一方，**(C)**，**(D)**，**(E)** は動的な相互関係を表しており，メッセージ送信で記述すべきものである．パターン **(C)**，**(D)**，**(E)** と OODJ の日本語単文の基本文型として定義した[8] S+V，S+O+V，S+C+V，S+O+C+V，S+O+O+V と比較すると，**(C)** と **(E)** は S+O+V に含まれ，**(D)** は S+O+O+V に含まれる．S は日本語単文では省略されることが多く，OO 表現の中では単文が記述されているシング（クラス）自体であること

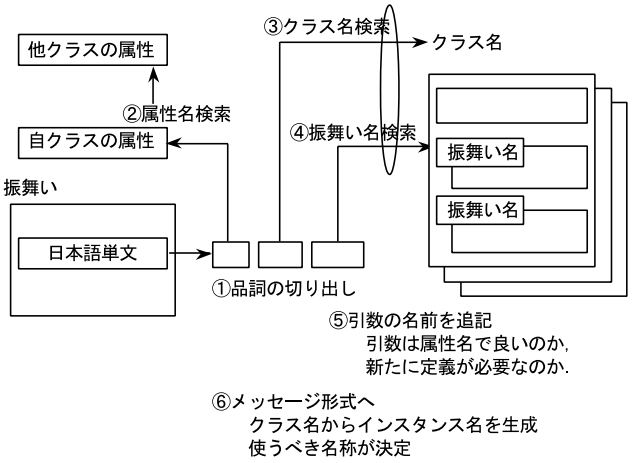


図 14 単文変換の手順
Fig. 14 Translation process of Japanese simple sentence.

①日本語単文の品詞分解
②品詞分解された文をオブジェクト部分（名詞：S）とメッセージ部分（動詞：V,目的語：0） に分割する.
③名詞：該当名が自身または上位のクラス名か属性名と一致するかを検索する.
上記になければ，該当クラス名をリポジトリ内の全クラスに検索する.
④動詞：③で見つけたクラスの中に該当する振舞い名を検索する.
該当振舞い名があれば，採用する.
なければ，新たなクラスを作成する.
⑤目的語：該当振る舞い名の引数との照合を行う.
照合の結果合わなければ，引数を補足する.
⑥クラス名からインスタンス名を生成する文を記述する．インスタンス名を③の名詞と置き換える.
⑦ ①から⑥を個々の文に対して行う.

図 15 変換プロセスの説明
Fig. 15 Translation processes.

(A) Haveパターン：（××は○○の部分である．）
オブジェクト間の全体・部分の関係を表す.
(B) Is-aパターン：（××は○○である．）
オブジェクト間の上位・下位の関係を表す.
(C) Giveパターン：（××へ△△を○○する．）
対象のオブジェクトに何かを与える.
一般にオブジェクトからの戻り値はない.
例：ディスプレイに結果を表示する.
(D) Takeパターン：（××から△△を○○する．）
対象のオブジェクトから何かを取り出す.
メッセージに対し戻り値が存在する.
例：電話帳からAさんの番号を見つける.
(E) 状態変化パターン：（××を○○する．）
対象オブジェクトの内部状態を変化させる.
例：電源のスイッチをオンにする.

図 16 単文のパターン文型の抽出
Fig. 16 Extraction of statement type patterns.

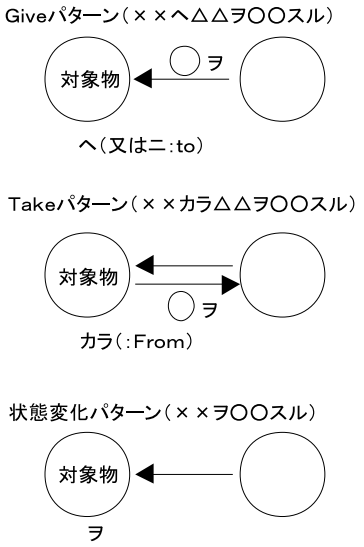


図 17 パターンの図式表現
Fig. 17 Syntax diagram of pattern.

が多い．一方，C の役割は補助的な意味合いが強く，メッセージ送信の引数に多くは変換できる．したがって，S+O+C+V は S+O+V の変形と見なすことができる.

S+V，S+C+V はシステムの用意する基本関数を利用するときのパターンであることが多い．**図 17** に**(C)**，**(D)**，**(E)** のパターンの図式表現を示す.

単文内部のパターン化は文章ではなく「テニオハ」のパターン化である．自然日本語は文法に定められた部分が元々少なく，記述に多様性が生じる可能性が高く，省略される場合も多く，ユーザが解釈しないと意

味が分からない場合も多い．したがって，ユーザが単文を上記パターンに意識的に書き直すことで，ユーザの意図する意味を引き出す役割も果たすことになる．

5.3 変換規則 1 の定式化

変換規則 1（図 4）は 5.1 および 5.2 節から次のようにまとめられる．

(1) パターン化（5.2）

オブジェクト指向記述への第 1 段階であり，メッセージ化への橋渡しとして日本語単文上でオブジェクト指向記述を意識した文章に書き直す．

(2) メッセージ化（5.1 の (a)）

パターン化した文章をメッセージ形式に変換．

オブジェクト ← メッセージ

パターンからメッセージ化するときに表現が変化することがあるのに注意．たとえば「電話帳から A さんの番号を見つける．」は

電話帳 ← 番号を検索せよ（A さん）．

引数の有無は出現箇所や内容によって決まる．

(3) 近似表現化（5.1 の (b)）

OOPD 文法に正確に従う記述へ移る前に，ユーザの自己流で OOPD に近い文（近似表現と呼ぶ）を記述するステップを設ける．これは論理的に手順をまず明確にすることを優先させるために設けた．論理的手順が明確になれば終了する．

また，ここでは厳密化（定量化）すなわち，曖昧な定性的表現を定量的表現への変換も行う．

- かなりの速度 → 250 km/h
- 比較表現の日本単語は記号に

  例：以上 → ＞＝

6. 支援環境の設計

6.1 ユーザとシステムの役割分担

OODJ 記述から OOPD 記述への変換を支援する環境の設計方針はユーザとシステムが役割を分担しながら変換を順次進める方式とする．システムの自動変換のみの方式では，たとえば，データ型等の自動変換の場合等でユーザの意図とシステムの生成した型が一致する保証はなく，間違った結果さえも導き，その修正も煩雑である．我々はこのことからシステムの自動変換ではなく，人間の適切な判断割込みを随時織り込んだ変換方式が望ましい形態である，と考えた．この方式をシステムと人との協調を意識して**自働変換**と名付け，支援環境の設計の基本方針とする．自働変換でのユーザとシステムの役割分担の切り分け基準として**図 18** の方針を設定した．ポイントは単語の意味解釈はユーザが行い，ユーザの判断材料はシステム側で提供する．具体的にはユーザは単語から意味に合ったクラスの選択や新規クラスの設計を行い，システムはユーザへクラス名や OOPD 文法の詳細を提示する．

役割分担の切り分け基準

(1) 単語の意味解釈はユーザ側で行う．
　OODJ単文からOOPD文への変換では，単語の意味によって既存のクラス，属性，振舞いの適切なものを選択する必要がある．ここにはユーザの知識，判断が介入する．ここでの中途半端な自動化は多くの混乱を招き，かえってプログラム作成を妨げる結果になる．
(2) ユーザの判断材料はシステム側で提供する．
　ユーザはOODJ で記述した対象世界のすべてを正確に記憶はしていない．従って，システム側が情報の検索・提示を行う．

具体的役割分担

ユーザの役割

(1) 単語の意味解釈と真の意図を抽出する．
　ここで抽出とは単語の意味を実現する他のクラスを見つける，あるいは新規クラスを設計することである．
(2) 意味に合うクラス 属性 振舞いを選択する．
(3) 意図をPLで表現する．
　例えば，属性へのデータ型の付加や条件判定の論理式などである．

システムの役割

(1) 抽出等を支援する（選択はユーザ）．
(2) PLの文法を誘導・提示する．
　ユーザのPLの知識は曖昧である．多くを提示し言語仕様のガイドを行う．記述箇所で使える表現を提示する．

図 18 ユーザとシステムの役割分担
Fig. 18 Role sharing between domain users and system.

(1) システム側は支援の立場に徹する．
(2) システムの支援機能の境界を明確にする．
(3) ユーザの入力する情報を蓄積し，他クラスの設計に役立つ情報とする．
(4) 出来る限りユーザ入力を少なくする．
(5) 最終的なOOPDプログラム記述はドメインユーザの責任とする．
(6) 振舞い記述の階層化はユーザ責任で行う．

図 19 支援環境の設計方針
Fig. 19 Design principles of support environments.

6.2 基 本 方 針

役割分担を基に支援環境の設計方針は**図 19** のように，システムの支援機能の境界を明確にし，一方でユーザの入力はできる限り少なくする等と決めた．

6.3 支援環境の設計

一般的にプログラマは日本語単文を元に（処理仕様書として解読），プログラムに書き直す．しかし，本変換システムはプログラマが行う頭の中でのこの種の変換を，できる限り顕在化する形で誘導する環境とし

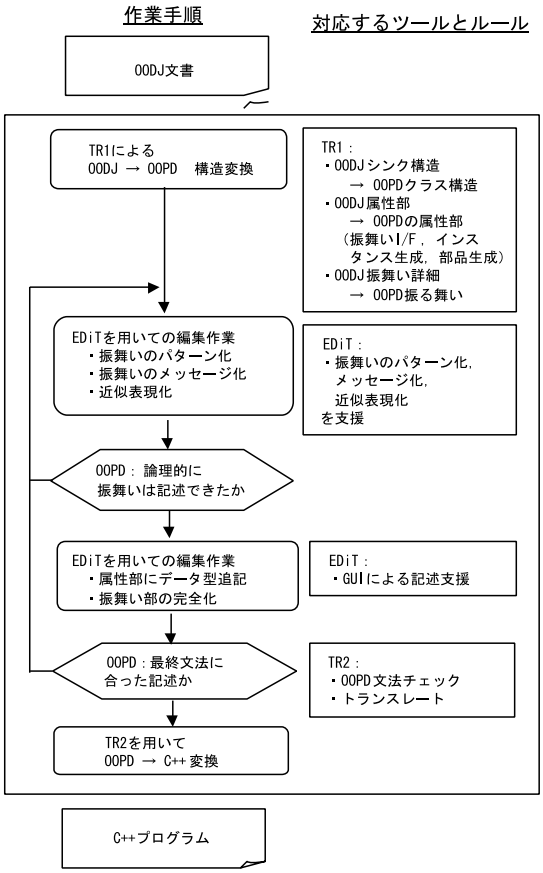


図 20 変換システムの概要
Fig. 20 Outline of translation system.

て提供する．したがって，プログラミングを不得意とするドメインユーザも，製作しようとするソフトウェアの全体構成を誤ることなく，局所的な部分に集中してプログラム化することができる．

図 20 に変換システムの概要を示す．

6.4 変換システム概要

図 20 に沿って変換システムの概要を述べる．なお，下記の**TR1**，**EDiT**，**TR2** システムは既存の日本語一貫プログラミング環境[6),7)] を基盤にして製作した．

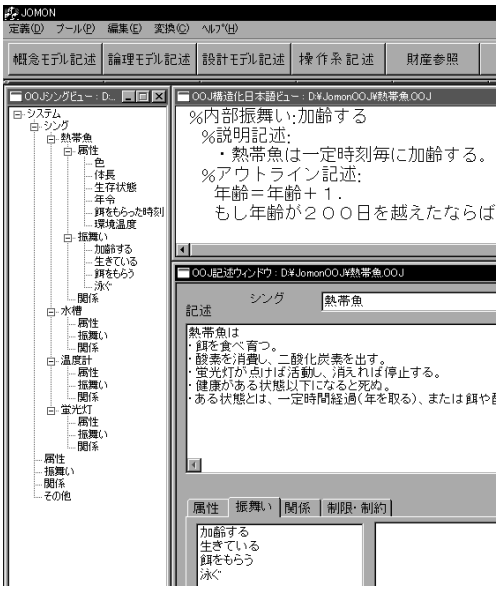
（1）入力データ（OODJ 文書）

OODJ 記述が終了すると**図 21** (a) のようになる．OODJ 記述文書は OODJ エディタによりリポジトリに格納される．これらはテキストとして格納されるが，同時に属性名や振舞い名は抽出されて辞書としても使える形式で格納され，OOPD 変換時にメッセージ送信文の送信先名の高速検索等に利用される．

（2）OOPD 概略変換とトランスレータ TR1

まず OODJ 文書を TR1 により OOPD へ概略変換する．TR1 は変換システムで最初に起動されるトランスレータで，OODJ 記述の全体の構造と属性部の変換を行う．OODJ のシング構造は OOPD のクラス構造へ，シング属性と振舞いは OOPD の属性と振舞い記述の定義部へほぼそのまま移行する．いずれもリポジトリに格納される．図 21 (b) に OODJ 文書が概略変換された結果の OOPD 画面を示す．

（3）編集作業とエディタ EDiT



(a) OODJ で記述した熱帯魚クラスの一部

(b) OOPD へ概略変換された熱帯魚クラス

図 21 OODJ と OOPD の画面
Fig. 21 OODJ and OOPD.

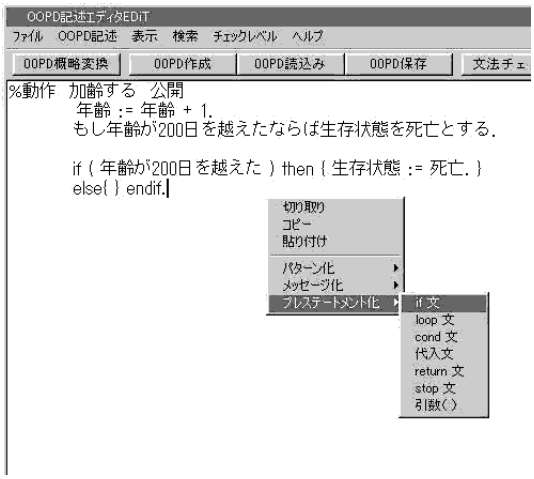


図 22 EDiT 使用画面
Fig. 22 EDiT in use.

ユーザはツール EDiT を用いて OOPD プログラムを記述する．EDiT は対話型変換を支援するエディタ（**図 22** 参照）である．ユーザは EDiT の機能を用いてパターン化，メッセージ化および近似表現化の記述を進め，OOPD プログラムの形の概略を作り上げる．論理的に OOPD プログラムが記述できるまで，詳細な文法にとらわれずこの作業を行う．システム側も（チェックレベル指定により）OOPD の文法チェックを厳密には行わない．

論理的に納得がいくまでこの作業を繰り返した後，ユーザは同じくこの EDiT を用いてデータ型記述追加等を行い，OOPD 文法にあったプログラムに書き換える．またリポジトリには対象世界の全クラスの情報が格納されており，ユーザが文中のオブジェクト部分を選択するとリポジトリ内の当該クラスおよび類似候補クラスが last recently 方式で表示される．

（4）TR2

OOPD 記述を C++プログラムへ変換する[6]．

7. 記述実験と評価

7.1 記 述 実 験

（1）実験内容

OOPD を用いて日本語一貫プログラミングの記述実験を行った．前論文の環境[6),7)] では，従来手法として OMT と C++の組合せを取り上げ，一貫プログラミングとの記述比較実験を行い，従来比開発効率 1.5 倍の効果が確認された．

本論文では前論文の環境と本論文での環境の相違を見る記述実験も行った．前環境と本環境との違いは日本語一貫プログラミングの書き換え手順の明確化と支援環境の充実である．

飼い主は
・１日１回　ある時刻に餌を与える。
・蛍光灯をつけたり消したりする。
・故障しないかを見張る。
水槽には
・水温を一定（２６度Ｃ前後）に保つためにサーモスタット付のヒーターを入れる。
・水温が低ければヒーターの故障なので取り替える。
・水温が高ければサーモスタットの故障なので取り替える。
・酸素が不足しないように空気ポンプを動かす。
・泡が出ていなければ空気ポンプの故障なので交換する。
・ヒーター、サーモスタット、空気ポンプの劣化は考えない（熱帯魚より長持ちする）。
熱帯魚は
・餌を食べ育つ。
・酸素を消費し、二酸化炭素を出す。
・蛍光灯が点けば活動し、消えれば停止する。
・健康がある状態以下になると死ぬ。
・ある状態とは、一定時間経過（年を取る）、または餌や酸素が一定時間与えられない場合。

図 23 概念モデルの記述
Fig. 23 Description of conceptual model.

%操作 飼い主の行動
%一時変数
%動作
飼い主＝シミュレーションをする人とする.
シミュレーションを起動する.
蛍光灯を点ける.
餌を何らかの方法で与える．しばらく眺める.
サーモスタット、ヒーター、蛍光灯の状態をチェックする.
温度計を見て２６度Ｃ以上ならばサーモスタットを交換する.
２６度Ｃ未満ならばヒーターを交換する.
蛍光灯を消す.
飼い主はシミュレーションを終了する.
STOP.

図 24 操作系
Fig. 24 Pattern-formed description of operation part.

（2）記述テーマ

本論文では OOPD の記述例としてモデル化が容易であること，相互作用が発生すること，および一般的で分かりやすいことから**図 23** に示す熱帯魚を飼育するシミュレーションを取り上げる．

7.2 記述サンプル

（1）記述分割とパターン化

図 24，**図 25**，**図 26** は OODJ で記述された論理モデルを OOPD 記述の第 1 ステップとして操作系とクラス定義系に分けて記述し，さらに操作系，クラス定義系の振舞い記述部分をパターン化したものである．パターン化はオブジェクトの操作に関わるので，操作

```
%外部仕様
%クラス　　熱帯魚
%クラス定義
　assoc　水槽　　中で泳ぐ
%属性　　色　　私有　カラー型
%属性　　体長　私有　長さ
%属性　　生存状態　　私有　　状態型
%属性　　年令　　私有　　整数型
%属性　　餌をもらった時刻　　私有　　時刻型
%動作　%説明　　；ＯＯＤＪで記述
　熱帯魚は一定時刻毎に加齢する．
　２００日経過すると死ぬ．
　餌をもらった現在時刻を記憶する．
　２日経過しても餌がもらえなかったならば，死ぬ．
　蛍光灯が点いているならば，移動する．
　蛍光灯が消えれば，止まる（見えなくなる）．
　温度が標準より一定時間はずれると死ぬ．
　　（条件を考える必要あり）．
―――――――――――――――――――――
%外部仕様
%クラス　　温度計
%クラス定義
　assoc　水槽
%属性　　温度　　私有　　整数型
%属性　　温度変化経過時間　　私有　整数型
%動作　%説明
　温度計は最初標準２６度Ｃとする．
　　（ヒーターで暖まっているとする）．
　サーモスタットが故障し、一定時間経過すると
　温度が上がる．
　ヒーターが故障すると、一定時間経過すると
　温度が下がる．
―――――――――――――――――――――
%外部仕様
その他のクラス（省略）
```

図 25 定義系（外部仕様）
Fig. 25 Pattern-formed description of definition (external specification).

```
%内部仕様
%クラス　　熱帯魚
%属性　　色　私有　カラー型　＝赤
%属性　　体長　私有　長さ　＝５ｃｍ
%属性　　生存状態　　私有　　状態型　＝０
　　（０：生存、１：死亡）
%属性　　年令　　私有　　整数型　　＝０
%属性　　餌をもらった時刻　　私有　時刻型　＝０
%動作　　表示する　　公開
　熱帯魚を表示する．
%動作　　生きている　　公開
　餌をもらった時刻を記憶し、２日経過しても
　　餌がもらえなかったら死ぬ．
　温度が標準より一定時間はずれると死ぬ．
　　（条件を考える必要あり）．
%動作　　加齢する　　公開
　熱帯魚は一定時刻毎に加齢する．
　２００日経過すると死ぬ．
%動作　　餌を与える　　公開
　餌をもらった時刻を記憶する．
%動作　　移動する　　公開
　蛍光灯が点いているときは動き、消えると止まる．
　水槽にぶつかる寸前に反転する．
%動作　　表示しない　　公開
　形状を消す．
―――――――――――――――――――――
%内部仕様
%クラス　　温度計

%属性　　温度　　私有　　整数型　＝標準温度
　　；標準温度＝２６
%属性　　温度変化経過時間　　私有　　整数型　＝０
　　；高温、低温共にこれを使う
%動作　　計測する　公開
　温度を戻す．
%動作　　標準に設定する　公開
　サーモスタット或いはヒーターを交換すると，
　　標準温度になる．
%動作　　温度が上がる　　公開
　サーモスタットが故障すると，温度が３０度Ｃに上がる．
%動作　　温度が下がる　　公開
　ヒーターが故障すると，温度が下がる．
%動作　　一定時間が経過した　　公開
　温度がずれて一定時間経過すると、熱帯魚を殺す．
```

図 26 定義系（内部仕様）
Fig. 26 Pattern-formed description of definition part (internal specification).

```
%操作　飼い主の行動
%一時変数
%動作
　シミュレーション←起動する．
　蛍光灯←点ける．
　熱帯魚←餌を与える．
　サーモスタット、ヒーター、空気ポンプ←チェックする．
　（「温度計」が２６度Ｃ以上）ならば、
　　サーモスタット←交換する．
　（「温度計」が２６度Ｃ未満）ならば、
　　ヒーター←交換する．
　（「空気ポンプ」の泡が出ている）ならば、
　　何もしない．　　交換する．
　蛍光灯←消す．
　飼い主はシミュレーション←終了する．
　STOP.
```

図 27 操作系：メッセージ化
Fig. 27 Message-formed description of operation part.

系に集中するのが一般的傾向である．

図 26 の内部仕様の動作記述（メソッド定義）は図 25 の外部仕様の説明を参照しながら書き進める．その過程で属性値等が追加になる．なお，属性の「私有」，動作の「公開」は〈アクセス制約〉を表し，システム側が既定値として設定してあるものを表示する．

（2）メッセージ化

図 27 に操作系の動作手順のメッセージ化を示す．クラス定義系では内部仕様の動作詳細化を行うが，本例では内部仕様のメッセージ化はあまりないので省略する．

（3）近似表現化

図 28，**図 29** に操作系とクラス定義の動作を近似表現化した記述を示す．

7.3 実験の評価

図 30 に実験の結果を記す．

（1）定量的評価

論理モデル化の記述は前回に比較し，OODJ の記述仕様[8] が明確になり，全体のクラス構造（OODJ ではシング構造）をこの時点で決定することになったこと，および単文で記述できるところまでできる限り詳細化したので，記述量，期間ともに増加した．記述行

```
%操作  飼い主の行動
%一時変数
%動作
;蛍光灯は消えている状態からスタートする.
;イベントドリブン・プログラミングに書きなおす.
;プログラミングの知識はサンプルを用意し，提示する.
 loop（永久）  {
   シミュレーション時計をカウントアップする.
   シミュレーション時計を表示する.
   イベントを取り込む.
   cond（キーの種類）  {
     case（Ｎキー）  {蛍光灯を点ける.
                      熱帯魚を表示する.}
     case（Ｆキー）  {蛍光灯を消す.}
     case（Ｅキー）  {熱帯魚に餌を与える（現在時刻 }
     case（Ｃキー）  {
       loop（次のキー押下まで）  {
         イベントを取り込む
         cond（キーの種類）  {
           case（Ｓキー）  {サーモスタットを交換する
           case（Ｈキー）  {ヒーターを交換する.}
           case（Ｐキー）  {空気ポンプを交換する.}
           default         {  }
         }
       }  //  endloop
     }
     default
   }
   if（熱帯魚は生存している）then
          {熱帯魚を加齢する．熱帯魚を表示する.}
     else      {シミュレーションを終了する.}
   endif.
 }  //  endloop
```

図 28　操作系：近似表現化
Fig. 28　Pre-statement description of operation part.

が 26 行と 62 行の 2 段階になっているのは論理モデル化の最初の記述と完成記述の両段階を示している．行数の増加はシングの追加と振舞いの詳細化である．

詳細モデル化の記述は，従来の JSMDL では 230 行の記述に 5 日間を要したものが，今回の OOPD では OODJ 段階でかなり振舞い記述が充実したので，2 日間と短期間で記述できた．パターン記述は OODJ 段階でかなり進んでおり，OOPD 段階ではいくつかの曖昧な文の発見と再記述があったのみである．メッセージ化，近似表現化では OODJ の振舞い記述がかなり詳細化されており，従来の JSMDL 記述に比べて記述がしやすかった．最終的な OOPD 記述は近似表現化までは 140 行であったものが，属性のデータ型，条件の詳細化，引数の設定等で行数は 260 行と膨らんだ．

モデル化の合計期間は，論理モデル化と詳細モデル化との間でクラス構造の決定や新規クラスの追加，振舞い記述の補足記述が発生し，モデル間を相互往来したので，従来の 6 日間が今回 5 日間であり，効率向上としては 20%程度にとどまった．テスト期間は OODJ と OOPD との相互往来により，アルゴリズムの机上デバックが進み実際のテストでは従来の 5 日間が 3 日間と短くて済むことができた．

```
%内部仕様     //  熱帯魚クラスの動作（下線部は動作名）
%動作  表示する      公開
         熱帯魚の形状を表示する.
              ;図形クラスを用いる（図形クラスは省略）
          if   （生存している）  then   移動する.
                 else   水槽の底に移動する.   endif.
%動作  生存している      公開
         if   （温度が30度Ｃ以上か20度Ｃ以下に
              なってから3時間経過した）   または
            （餌をもらった時刻から2日間経過した）
               then     生存状態：＝  死亡.   endif.
%動作  加齢する      公開
         年齢：＝  年齢  ＋  １.
          if（年齢が200日を越えた）then
                    生存状態：＝死亡.   endif.
%動作  餌を与える（現在時刻）      公開
          餌をもらった時刻：＝  現在時刻.
%動作  移動する      非公開
                    ;表示するから派生して作られた
          if（水槽の範囲を超えた）then
                    方向を反転する
           else  熱帯魚の形状を移動する.  endif.
%動作  水槽の底に移動する      非公開
                    ;表示するから派生して作られた
          熱帯魚の形状を水槽の底に移動する.
%動作  表示しない      公開
          熱帯魚の形状を消す.

温度計クラスの動作
%動作  計測する      公開
          return（温度）.
%動作  標準温度にする      公開
          温度：＝標準温度.
          温度変化経過時間：＝０.
%動作  温度が上がる      公開
          温度：＝高温.          ；高温＝３０
          温度変化経過時間：＝１.
%動作  温度が下がる      公開
          温度：＝低温.          ；低温＝２０
          温度変化経過時間：＝１.
%動作  一定時間が経過した      公開
          if   (温度変化経過時間が０でない）then
            温度変化経過時間：＝温度変化経過時間＋１
          endif.
```

図 29　定義系：動作部分の近似表現化
Fig. 29　Pre-statement description of definition part (behavior).

開発段階	比較項目	従来の支援環境	今回改訂の環境
概念モデル化	記述量	日本語 12行	日本語 12行
論理モデル化	記述量	OODJ 26行 （自由記述に近い）	OODJ 26行
			OODJ 62行
	期間	1日間	3日間
詳細モデル化	記述量	JSMDL 230行 （生成 780行）	近似表現化まで 近似表現 140行
			OOPD 260行 （生成 810行）
	期間	5日間	2日間
プログラム化 ～テスト	期間	5日間	3日間
合計	期間	11日間	8日間

図 30　実験結果
Fig. 30　Result of description experiment.

全過程を通じて従来の環境と今回改訂の環境を用いての効率向上は 37%となった．また，プログラミングの学習期間はほとんど必要がなく，エディタの使用法について合計で数時間程度を要したのみである．

（2）定性的評価

記述実験に参加した数名の被験者の意見として，次の点があげられた．

単文のパターン化は不足する主語や対象が明確になり，操作系の記述で効果があった．定義系の記述ではメソッド内の振舞い記述部分は少ステップであり，単文のパターン化よりは選択文や代入文の出現率が高かった．メッセージ化は日本語の文章からオブジェクト指向プログラムへの思考パターンの切替えになり，オブジェクト指向初心者におおいに役立った．また，近似表現化は近似表現記述の後の OOPD 文法チェックで構文エラーが多発するが，はじめに論理の展開に集中し，記述できることは，構文エラーにとらわれることなくまず論理の記述ができる（してよい）という心理的な効果が大きく，評価できる．

一方，完成した OOPD 記述全般に対して，第 3 者（実験非参加者）による可読性の評価を行ったところ，「分かりやすい」記述という意見が多かった．理由としては，論理モデルの記述からの書き換えであるために OODJ の日本語単文が OOPD の動作記述に識別子等で残っていることと，概念モデルの日本語が説明文として OOPD にもコメントとして残るために，当初の意図が分かりやすいとの評価であった．さらに，操作系とクラス定義系，外部仕様と内部仕様の記述を分離したことが全体の設計見通しを良くし，可読性の向上にもつながっているという意見が寄せられた．

8. 評　　価

8.1 OOPD 文法の評価

OOPD 文法は次の 2 点から評価できる．

（1）高可読性と実行可能性の両立

OOPD の動作記述は OODJ 日本語単文記述の表現をそのまま利用，書き直しており，日本語一貫プログラミングであるがために高い可読性を維持できている．一方，書き直された OOPD の動作記述は元々実行可能であった JSMDL の動作記述にあてはまり，実行可能性を実現できている．

（2）設計仕様記述のガイド

プログラム設計仕様書の記述形式を言語仕様に取り込むことで，ドメインユーザの設計記述のガイドを実現できた．

JSMDL は単に 1 つのクラスを属性と振舞いに分けて記述するプログラム記述言語であった．OOPD は外部仕様と内部仕様記述との分離で，一般的な設計記述手法を文法の中に取り込むことと，操作系と定義系との分離でトップダウン記述とボトムアップ記述を明確にすることにより，ドメインユーザへの記述ガイドを実現できた．

いい換えれば，従来は論理モデルに現れたシングを詳細モデルのクラスへ書き直すだけであったが，設計仕様記述を取り込むことで，設計全体の見通しが良くなったといえる．

8.2 変換規則と支援システムの評価

変換規則 1 はメッセージ化等の一般によく用いられる手法を記述手順として組み合わせ，OODJ の可読性の高い状態を保持したままで実行可能な文へ円滑に変換する変換プロセスを確立したもので，支援環境として実現できた．また，この支援環境はクラス構造の穴埋め式記述，基本データ型のメニュー選択，定義側（インタフェース）を参照しながらの振舞い記述等の多彩な入力形式を用意することで，初心者から慣れたユーザまでストレスを感じない入力量となっており，**7.1** 節の記述実験においても被試験者の評価は高い．

一方，変換規則 2 の実装であるトランスレータは OOPD 文法検査においてエラーメッセージの内容がまだ適切でないところが多く，改良の余地ありとの評価であった．

8.3 プログラミング学習フリーの実現

OOPD はプログラミング一歩前のプログラム設計段階の記述言語であり，ドメインユーザに要求されるプログラム記述の知識は，オブジェクト指向設計の基礎（クラスの静的構成，カプセル化概念，クラス継承，属性と変数の関係，メッセージ送信による振舞いの記述），少々のソフトウェア設計と Fortran，BASIC 程度のプログラミングの経験，および OOPL のイメージである．

上記の経験を持つユーザが，OOPD を容易に使えるかどうかは**7.1** 節の評価実験で，数時間のエディタ使用法の学習で OOPD が使えるようになったことから，実現できたといえる．学習容易性の理由としては，実行効率を上げる文や物理構成を意識させるプレ記述機能を省いたことがあげられる．実行効率を上げる機能はプログラミング学習の負担を重くする傾向にある．OOPD は設計記述言語であるという観点からこれらの機能を文法から省いたことが学習しやすくなった原因と考えられる．

なお，OOPD トランスレータでは英小文字の文字列を透過する機能があり，実行効率を上げたいプログ

ラマが C++を意識的に OOPD の中に混入することは可能になっている．

8.4 比較検討

ここでは，OOPD の位置付けおよび特長において対比関係にあるものと比較検討する．

本研究が再構築した OOPD はプログラム設計工程の仕様をオブジェクト指向に基づいた概念で記述できる言語である．これまでの多くのオブジェクト指向開発法では，分析・設計工程の仕様作成に重きが置かれており，実装工程のオブジェクト指向言語との結び付きとなるプログラム設計仕様の文書化は示されていない．一般的には，オブジェクト指向開発法とプロトタイピング手法のつながりで利用されるため，なおさらプログラム設計の意義が薄れているといえる．また，現実的にはオブジェクト指向開発のエキスパートによるプログラム開発を前提としており，プログラム設計の明文化を必要とせずに特定のプログラミング言語で実装を進めることができる．しかし，本研究が想定するドメインユーザにはそのようなことは無理であり，オブジェクト指向プログラミングよりも，プログラム設計に専念し，自動変換による開発とした方が，より良いプログラムを確実に開発できると考える．この観点において，自然言語風のプログラム設計仕様記述言語の提供は意義あるものといえる．

プログラム設計記述言語に位置付けられるものとして，第 4 世代言語[3] をあげることができる．しかし，これはビジネス・基幹システム記述向けであり，科学技術計算には向いていない．

オブジェクト指向において分析・設計とプログラミングを結び付けるものとして，デザインパターン[10] が存在するが，これはオブジェクト指向開発エキスパートには使いこなせるが，我々の対象とするドメインユーザにはモデルとなるクラス構成が理解しにくく利用困難なものである．

また開発を支援するものとして，Rational Rose[11] をはじめとする多くの CASE ツールが存在するが，基盤となる UML 等の仕様記述モデルは構文上の正しさが求められており，ドメインユーザが使うには煩雑すぎるといえる．実装言語への移行を支援する機能もあるが，やはり C++のようなプログラム言語を用いての直接のプログラミングから逃れることはできない．

OOPD のさらなる特長として文書化されたプログラム設計仕様の理解容易性や高可読性をあげることができる．COBOL や第 4 世代言語はプログラミング言語でありながら省略表現を多用しないことから文書としての可読性は比較的高いといえる．また，Eiffel[12] はオブジェクト指向プログラミング言語でありながらプログラムの文書化を意識した言語仕様となっている．これらの言語記述に対し，OOPD は設計仕様記述段階での文書として読むことができ，明らかに可読性は高く，理解容易性に優れているといえる．

9. 結論と今後の課題

本論文の核心はプログラム設計記述言語 OOPD の設計とその記述支援環境の構築である．OOPD は自然言語の能力を保持する高可読性の記述言語であると同時に，実行可能な記述を実現した記述言語である．この実現のために，従来の日本語一貫プログラミングで用いたプログラム言語 JSMDL の改訂，分析・設計を行うための記述言語 OODJ からこの OOPD への変換規則の定式化を行った．

JSMDL から OOPD への改訂では記述モデルの明確化とモデルに基づく文法定義を行った．文法をプログラム記述言語からプログラム設計言語へと上流工程へ移行し，実行効率等を一部犠牲にし，可読性の向上と学習容易性を中心にする文法とした．

また，変換規則は変換プロセスを解析し，そこにパターン化，メッセージ化，近似表現化の手順を見いだし定式化することができた．この結果を，自然日本語，OODJ，そして OOPD の系統的記述を支援する日本語一貫プログラミング環境へ反映し，専門領域の知識は豊かであるが，プログラミングを不得手とするドメインユーザに対し，OOPD 段階でプログラムを製作できるに十分な開発環境を構築することができた．

今後の課題は OODJ から OOPD への変換規則をよりいっそう自動化することである．

謝辞 本論文作成にあたり，記述環境構築に協力された茨城大学大学院理工学研究科院生の永松泰成君に謝意を表する．

参 考 文 献

1) 今城：日本語プログラム文献ノート，情報処理学会秋のプログラムシンポジウム「日本のプログラミング」報告集，1998-9-16, p.18 (1999).
2) 今城：日本語プログラム言語の設計，情報処理学会冬のプログラムシンポジウム報告集，p.36 (1999).
3) 吉野，田村，稲益：ソフトウェア開発支援ツール “SEWB3，EAGLE/4GL” の機能と特長，日立評論，Vol.75, No.11, pp.21–28 (1988).
4) 佐川，金野，梅谷：数値シミュレーション言語 DEQSOL，情報処理学会論文誌，Vol.30, No.1, pp.36–45 (1989).
5) 廣田，橋本，長澤：応用ドメインに特化した概

念モデル記述言語に関する一考察，情報処理学会論文誌，Vol.36, No.5, pp.1151–1161 (1995).
6) 加藤木，畠山：オブジェクト指向日本語一貫プログラミング環境，情報処理学会第 118 回ソフトウェア工学研究会，98-SE-118, pp.15–22 (1998).
7) 加藤木，畠山：オブジェクト指向日本語一貫プログラミング環境，情報処理学会論文誌，Vol.40, No.7, pp.3016–3030 (1999).
8) 畠山，加藤木，石井：オブジェクト指向記述日本語 OODJ とその記述環境，情報処理学会論文誌，Vol.41, No.9, pp.2567–2582 (2000).
9) 加藤木，畠山，小林：シミュレーション向けのオブジェクト生成支援環境の設計と実装，オブジェクト指向 '95 シンポジウム，情報処理学会シンポジウム論文集，Vol.95, No.3, pp.205–212 (1996).
10) Gamma, E., et al.: *Design Pattern*, Addison-Wesley (1995)，本位田，吉田（監訳）：デザインパターン，ソフトバンク.
11) Rational Software Corp.: Rational Rose 98 (1998).
12) Meyer, B.: *Object-Oriented Software Construction*, Prentice-Hall (1989). 二木（監訳），酒匂（訳）：オブジェクト指向入門，アスキー.
13) 畠山，加藤木，上田：オブジェクト指向自然日本語 OONJ の設計と評価，情報処理学会第 130 回ソフトウェア工学研究会研究報告（Mar. 2001）.

(平成 13 年 2 月 27 日受付)
(平成 14 年 2 月 13 日採録)

加藤木和夫（正会員）

1970 年北海道大学理学部物理学科卒業.（株）日立情報制御システムを経て，2002 年 4 月より加藤木技術士事務所．制御用プログラミング言語，ソフトウェア開発環境および情報システム等の研究・開発に従事．著書「PAD によるプログラム技法」（共著），「Smalltalk/V によるオブジェクト指向プログラミング」ほか．技術士（情報工学部門）．電子情報通信学会会員.

畠山　正行（正会員）

1976 年東京大学大学院博士課程（航空学専攻）修了．工学博士．東京都立航空高等専門学校を経て，1990 年 10 月より，茨城大学工学部情報工学科講師，同助教授を経て現在に至る．この間，超音速凝縮流れ等の非連続気体流れの数値的な解析の研究に従事．次いで，数値シミュレーションユーザ用の開発および実行環境，オブジェクト指向に基づくファイルシステム，分散・並列システムとそのシミュレーションシステムや手法の研究開発に従事．日本機械学会，日本航空宇宙学会，日本数値流体力学会，日本計算工学会，日本シミュレーション学会，電子情報通信学会，ACM 各会員.

上田　賀一（正会員）

1961 年生．1989 年名古屋工業大学大学院工学研究科電気情報工学専攻博士後期課程修了．同年同大学工学部電気情報工学科助手．1990 年茨城大学工学部情報工学科講師，現在に至る．ソフトウェア工学，プログラミング言語に関する研究に従事．工学博士．電子情報通信学会，日本ソフトウェア科学会，ACM，IEEE Computer Society 各会員.